Bell Laboratories

subject: C Language Portability

date: September 22, 1977

from: B. A. Tague

While it is true that C is a portable language, merely using C is no guarantee of portability.

Portability usually refers to the ease with which a particular program can be made to run correctly on another computer system (i.e., different hardware and operating system). However, to the BTL C programming community, a more common portability frustration involves simply moving programs between UNIX variants (including MERT) or between releases of the C compiler.

In the first sense, the portability of C programs have been recently enhanced by new tools developed by center 127:

- Additions have been made to C data specification facilities.

- The Standard I/O package implements efficient portable I/O. This package should be used for nearly all new C development. The bulk of existing programs should be converted, as well.

- Standard header files have been developed to describe common system-dependent data structures. These declarations should be used in existing and future programs.

- The lint program (a form of C verifier) is available to flag portability problems.

Department 8234 is committed to promote portability in the following ways:

- During 1978, the various UNIX variants and MERT will be replaced by the UNIX/TS and UNIX/RT offerings (both based on Research Version 7 UNIX). TS and RT will be kept as compatible as possible.

- C compiler releases will be issued approximately quarterly instead of annually. Experience shows that programmers want new features (or programs that use them), and that more frequent incremental changes are preferable to a significant annual change.

- Conversion of existing programs (or programs written between now and mid-´78) will be mildly painful. We recommend that conversion be started now using the Version 7 conversion routines contained in the ``-17´´ library which was issued as UNIX Mod. 3.0a.

CD-UNNL-1

MELCHNER,MELVIN J
MH2D518
SUBJECT MATCH

- Finally, the attached memoranda by R. C. Haight, A. L. Glasser, and T. L. Lyon expand on these points and offer guidance on writing portable C. The content of these separate documents will be merged and expanded upon as time and experience dictate.

Since neither UNIX nor C is ``finished'', the items above should be carefully considered. Obviously, this is a stop-gap publication. We shall improve it in the future.

Berkley A Tague

B. A. Tague

Copy to
M. L. Almquist, Jr.
T. F. Arnold
E. Arthurs
G. L. Baldwin
D. L. Bayer
D. Bergland
A. P. Boysen, Jr.
M. M. Buchner
J. G. Campagna
G. Chesson
R. W. Downing
M. P. Fabisch
A. Feiner
R. J. Griffith
R. C. Haight
B. H. Heider
E. A. Irland
M. M. Irvine
S. C. Johnson
J. R. Kane
H. P. Kelley
G. W. R. Luderer
H. Lycklama
J. F. Maranzano
N. A. Martellotto
R. W. Mitze
E. N. Pinson
T. M. Raleigh
J. Sheehan
W. B. Smith
R. Staehler
R. J. Watters

Bell Laboratories

subject: C Language Portability

date: September 22, 1977

from: R. C. Haight
MH 8234
2F208 x7498

PROGRAMMER'S NOTES

C is a portable language. At this moment there are C compilers for eight to ten different machines. However, the portability of the compiler does not automatically descend to programs written in C. C is first and foremost a systems implementation language. As such, it offers the power to refer to bytes within words, bits within bytes, and to actual memory locations. C must also remain expressive and efficient. Where other portable languages have taken a Victorian view of the unmentionable specifics of a particular machine, C provides for frank description.

There are several aspects of C portability -- portability to different hardware, to a different operating system, to both of the above, and to a UNIX variant (i.e., an earlier or modified version). These points are discussed below.

1. Hardware problems

The attached draft memorandum by T. L. Lyon faithfully describes his work in moving UNIX commands written in C from the PDP-11 to the Interdata. His finding is that correct use of new C features, header files, and the Standard I/O package served their purpose in solving most hardware portability problems.

Also of interest is R. W. Mitze's memorandum (MF 1273-770907) on the PDP-11 byte order problem.

2. Running programs developed on UNIX under another operating system

The main problem areas are:

- The difficulty of making most I/O systems simulate UNIX files. Reasonable equivalents of seek are especially hard to come by.
- Lack of facilities to create and manage processes, or if that is possible, lack of means for testing the exit status of child processes. (These problems prevent the UNIX Shell from being `ported,´ and the Shell is called by many existing programs.)

- The pipe system call is not available or cannot be made useful, for either of the previous reasons.

- On IBM, EBCDIC character set problem complicates the operating system interface (The EBCDIC machine might be treated as ASCII if it weren´t for the need to pass character strings to the operating system).

3. Moving programs to another UNIX (or MERT)

The problems are:

- Trying to compile a new C program with an old compiler (The answer is stay up to date.)

- In the past, explicit file system path names have been built into many programs (i.e., names of files, devices, or commands). Restraint is called for. (A different kind of file linking is coming, I believe.)

- If a C program uses facilities peculiar to some UNIX variant, portability will be diminished.

- The transition from Research Version 6 (includes PWB, USG 3, etc.) to Research Version 7 (including UNIX/TS and UNIX/RT) is not bad for individual programs. However, all programs should be recompiled, and many will require source changes. A special subroutine library (`-l7´) permits programs to be converted in advance of Version 7 installation.

4. How to write portable C

The list below is admittedly sketchy (and aimed at C programmers).

- The typedef facility should be used `hide´ type-specifiers that are likely to vary from machine-to-machine -- a single change fixes all declarations.

- Type-specifiers should be chosen with care. If the range of a variable may exceed 16 bits, declare it as long. On the other hand, just because 16 bits is enough is not a sufficient reason to declare it to be short. Short arrays save space, but arithmetic on short variables may be less efficient than for int type on some machines. Tread the middle ground.

- The sizeof operator yields the width (in bytes) of its operand. Used with the type-name feature, it returns the size of basic data-types -- `sizeof( int * )´ is `2´ on the PDP-11, but is `4´ on the Interdata 8/32.

- The PDP-11 `signed character´ can be handled by masking (as in `c & 0377´).

- The functions isalpha(), isdigit(), etc., found in the `-lS´ library provide portable character-class tests.

- Bit field declarations are valuable, but can lead to portability problems. As long as you are only interested in one field at a time and do not count on any particular left or right positioning, bit fields can be safely used for packing information. In C, bit fields are assigned in the inherent byte order of the machine (right to left on the PDP-11, left to right on other machines). For all of the possible problems, bit fields are more portable than previous `mask and shift´ operations.

- A <u>union</u> involving bit fields and regular <u>type</u>-<u>specifiers</u> is unlikely to be portable (on the PDP-11 bit fields are assigned from the right).

- Use constructs like `x & ~077´ in place of `x & 0177700´ to be independent of word size.

- The <u>union</u> declaration allows for honest overlay definition. Use it.

- If you need to do mixed arithmetic on integers and pointers, be open about it; use <u>type</u>-<u>names</u>.

- Watch out for differences between `0´, `(long) 0´, and `(thing *) 0´.

- Finally, when you must write something that is machine dependent, make it a macro (the underside of `Top Down´).

5. <u>Short-term advice</u>

We recommend that the following actions be taken immediately:

- Most I/O in existing programs should be revised to use the standard I/O library. Source duplications of these functions should be removed.

- All existing programs that use the stat(), fstat(), or seek() should be changed to use the version 7 conversion library routines (New programs too). The version 7 header file `stat.h´ must be used as well. (The main problem with version 7 stat() is the regular file flag.) See the attachment for a list of useful routines and header files.

- Programs should be converted to use the standard header files kept in the /usr/include directory.

- The <u>lint</u> program (a form of C verifier) should be used to check C source programs. Both portability and strict-type-checking messages should be largely cleaned up. This program is a valuable aid. I have actually found serious bugs with it.

- Most constants found in C source are suspect. Those that can be replaced by the <u>sizeof</u> operator should be immediately eliminated. Certain others should be moved to the beginning of the source as <u>define</u> statements (or to header files).

To follow this advice, you will have to make fairly extensive (though mechanical) source changes. As pointed out in the Lyon memo, source programs will get smaller and object programs will get larger (speed is probably a standoff). Long-term maintainability will be greatly enhanced. Making these changes now will be painful, but putting it off will only make it worse.

6. Long-term advice

- Don't suffer from C-lag. Keep up to date with compiler releases. Remember, even if there are problems, it's true that "misery loves company." Depatrment 8234 plans to issue a C compiler update at the beginning is 1978. An up-to-date set of C support facilities will be included (run-time libraries, header files, and manual pages). The version 7 conversion routines will be part of the regular C library.

- When UNIX/TS and UNIX/RT are issued in mid-'78, convert! If you have followed the advice above, the impact will be minimized. Remember, enhancements will be made only to TS and RT.

. . .

Program understandability is another aspect of portability/maintainability (if the new programmer can't understand the program, it will be rewritten). Stick to normal C programming style. It may not be your style, but use it anyway, or your work will be reviled. If you have questions about style, Alan Glasser's "A Style Sheet for C Programming" is a good start (Memorandum for File dated February 28, 1977).

Much of this material was supplied by S. C. Johnson. D. M. Ritchie, T. L. Lyon, J. R. Mashey, and R. W. Mitze have also contributed.

RC Haight

R. C. Haight

Attachment
Library Routines and Header Files
MF by A. L. Glasser dated 2/28/77 "A Style Sheet for C Programming, Issue 1"
Paper by T. L. Lyon "Inter-UNIX Portability"

Copy to

Attachment I

Library Routines

Starred items below are available from the USG `stockroom´. The rest were sent out with the latest C compiler `mod´. All of these (with manual pages) will be re-issued with the next C compiler release.

fstat(file descriptor, stat buffer)
Returns the status information for the open file defined by the file descriptor. This function should be used with the header file stat.h.

isatty(file descriptor) *
Returns 1 if the file descriptor refers to a "typewriter" device.

seek(file descriptor, (long)offset, direction)
Moves the file pointer to the offset/direction given. Direction has three valid values: `0´) move pointer to offset, `1´) set pointer to the current location plus offset, and `2´) set the pointer to the size of the file plus (possibly negative) offset. Remember, files can be mangled by forgetting the long middle argument. In the `-l7´ library, the routine was called `lseek´ (it is even better to use the lseek() in with the standard I/O package).

stat(file name, stat buffer)
Returns the status information for the file named. Use stat.h with stat(). Where appropriate, the access() call is simpler and more portable.

char *ttyname(file descriptor) *
Returns a pointer to the path name of the typewriter (or zero on failure).

Header Files

ctype.h -- character class macros
Used with the standard I/O character class tests.

dir.h -- structure of a directory entry
Use when reading directory files.

sgtty.h -- stty(), gtty() structure
A very machine dependent structure.

signal.h -- definition of the signal() flags *
Use in place of your own constants or defines.

stat.h -- structure returned by the (version 7) stat()
Discard current source.

stat6.h -- version 6 stat() structure

stdio.h -- definitions used by the standard I/O package
The standard I/O package is very valuable in terms of both portability and long-term maintainability.

time.h -- structure returned by localtime() and gmtime() *

type.h -- portable `typedef´s

ATTACHMENT II   Bell Laboratories

subject: A Style Sheet for C Programming, Issue 1
Case: 39380
File: 39380-48

date: February 28, 1977

from: Alan L. Glasser
PY 9144
1A-117 x7922

MEMORANDUM FOR FILE

1. INTRODUCTION

The "style sheet" which appears below describes the recommended style (physical layout and appearance) for writing C programs in Center 914. Comments are hereby solicited from interested readers. As indicated by the title, it is expected that this document will be reissued as a result of readers´ comments.

This style sheet is derived from existing practice (e.g., the source of UNIX and the C compiler). It is not a "sudden or ill-considered" standard. There can sometimes exist good reasons for differing from these recommendations, although these should be few and far between.

It is hoped that these recommendations are viewed as being reasonable and simple to follow.

There are, of course, omissions in this document. The only style considerations considered are those which are generally accepted and used, and those which pertain to often used constructs. The experienced C programmer should be able to extend the style for situations not covered. Inexperienced C programmers should consult (1) existing C programs which tend to follow this style (one should consult a local UNIX expert to find out where to find C source code), or (2) experienced C programmers, in that order. The author does not have any strong opinions on the style to be used for omitted constructs.

If everyone writes programs in the same style, then everyone will have a simpler time reading those programs. Beautifiers are not an acceptable solution. Given three programmers each with a unique style, then in order for the first programmer to have a simple time reading the programs of the other two programmers there needs to be two beautifiers to convert the style of the second and third programmers to that of the first. It is much simpler for everyone to simply write in the same style.

The attached program listing is an example of the use of this style in practice.

2. COMMENTS

The first thing in a C program file should be a comment describing what the code in the file is all about.

```
	/*
	 *	This is a
	 *	sample comment.
	 */
```

The "/*" and the following "*/" stand on lines by themselves. Also, the "/*" begins at the left margin, and the text is indented one tab. Many people prefer to highlight comments (especially important comments) with some easily noticeable string down the left margin. Note that a "grep '^.*' files.c" will extract all comments from a file in which comment lines always begin "/*" or " *". ?

3. INCLUDES AND DEFINE CONTROL LINES

Following the comment should be any include or define control lines:

```
	# include	<stdio.h>

	# define DEBUG(V,T)	printf("V = %T\n", V)
	# define getchar()	getc(stdin)
```

Note the use of blanks and tabs. Note also the use of upper case letters in defines. All define variables, except for macros which could otherwise be implemented as subroutine calls, should be upper case. Subroutine names, and macros which may replace them, should be lower case.

4. STRUCTURE DECLARATIONS

The following is an example of a structure declaration:

```
	/*
	 *	Structure of a directory entry.
	 */

	struct	dir {
		int	d_ino;
		char	d_name[14];
	};
```

Note the comment preceding the structure (for more complicated and/or esoteric structures there should be a comment for each member of the structure). Note also that each member of the structure begins with the first character of the structure tag, followed by an underbar (_). This helps to identify member-of-structures in expressions. When there are two or more structures with the same first letter, use the first <u>two</u> letters of the structure tag (or, if possible, a different, meaningful, and unique single letter). Finally, note the use of blanks and tabs.

5. GLOBAL VARIABLES

When a single program is made up of many files, or the single file is very large, it is helpful to begin global variables with an upper case letter to distinguish them from local variables:

```
        char    External[]      "This is a global variable declaration";
        int     Tty[3];
        static  char    Sccsid[]        "%W%";
        char    Strings[]       {
                "first string",
                "second string",
                "third string"
        };
```

Note that the declaration begins at the left margin; also note the use of tabs. The last declaration above is the way in which SCCS identification information should be specified. This declaration should be as close to the beginning of the file as possible.

6. SCCS FILE IDENTIFICATION

It is also extremely useful to use SCCS identification keywords in the comment at the beginning of the file. A typical form might be:

```
        %M% %I% of %G%
```

After preprocessor statements, structure declarations and global variable declarations come the subroutines themselves.

7. SUBROUTINE DECLARATIONS

Subroutine names, their arguments, and local variables should not contain any upper case letters. The subroutine name should begin at the left margin (note that this allows the use of "/^function-name/" in the editor); any type declarations should precede the line that contains the subroutine name:

```
        long
        atol(string)
        char *string;
        {
                Body of function.
        }


        char *
        calloc(num, size)
        int num, size;
        {
                Body of function.
        }
```

The argument declarations should begin at the left margin. The open brace ("{")

for the function stands alone at the left margin on the line immediately following the argument declarations. This style allows the use of "?^{?,/^}/p" in the editor to print the function. (Note that this placement of the open brace is the only exception to the rule of placing open braces on the line that precedes the compound statement.)

A comment describing what the subroutine does should precede each subroutine.

Two blank lines (preceding any preface comment) are used to separate consecutive subroutines within the file.

8. WITHIN FUNCTIONS

Follow local variable declarations with a blank line. In general, a blank should follow a keyword whenever a parenthesis follows the keyword. The statement following an if, while, do, else, etc. should begin on the line following the control keyword, and be indented one tab from it. When compound statements are used within functions, the open brace ("{") should be on the same line as the control keyword which precedes the compound statement. The close brace ("}") should be on a line by itself, and line up with the statement that ends with the corresponding open brace.

The case statement should line up with the switch that goes with it. The statements within a case should be indented one tab from the case (and should be on separate lines!).

```
        switch (expres) {
        case CONS1:
                action;
                   .
                   .
                break;
        case CONS2:
                action;
                   .
                   .
                break;
        default:
                action;
                   .
                   .
                break;
        }
```

A do-while should look like:

```
        do {
                stuff;
                   .
                   .
        } while (expr);
```

Blanks after commas are useful. Blanks should always be used around assignment operators, conditional operators, and other binary operators (not primary expression operators, however -- see the syntax summary, appendix I of the C Reference Manual, D. M. Ritchie, TM-74-1273-1). Never put a blank between a unary operator and its operand.

9. FINAL REMARK

It should be noted that the most important recommendations to follow are the indenting conventions and the use of "white space".

Alan L. Glasser

Alan L. Glasser

PY-9144-ALG

Copy (with att.) to
A. Barofsky
D. L. Bayer
E. W. Boehm
L. E. Bonanni
A. P. Boysen, Jr.
K. J. Busch
R. H. Canaday
I. A. Cermak
N. F. Chaffee
M. B. Chasek
R. R. Conners
T. A. Dolotta
M. P. Fabisch
V. J. Fortney
H. S. Gellis
H. T. Gibson
R. F. Graveman
P. V. Guidi
R. C. Haight
C. B. Hergenhan
E. L. Ivie
M. M. Kaplan
B. W. Kernighan
D. B. Knudsen
J. S. Licwinko
T. G. Lyons
J. F. Maranzano
J. R. Mashey
M. D. McIlroy
R. E. Menninger
T. B. Muenzer
M. J. Petrella
M. A. Pilla
M. F. Pucci
G. W. R. Luderer
D. M. Ritchie

Continued next page

Copy (with att.) to - cont´d
M. J. Rochkind
M. M. Rochkind
W. D. Roome
R. F. Rosin
A. L. Sabsevitz
L. R. Satz
D. W. Smith
L. I. Stukas
B. A. Tague
P. D. Wandzilak
L. A. Wehr
D. E. Woolley

```
# include       <stdio.h>
# include       <macros.h>
# include       <fatal.h>

# define KEYSIZE        50

SCCSID(%W%);

/*
 *      Program to locate helpful info in an ascii file.
 *      The program accepts a variable number of arguments.
 *
 *      The file to be searched is determined from the argument. If the
 *      argument does not contain numerics, the search
 *      will be attempted on '/usr/lib/help/cmds', with the search key
 *      being the whole argument.
 *      If the argument begins with non-numerics but contains
 *      numerics (e.g, zz32) the search will be attempted on
 *      '/usr/lib/help/<non-numeric prefix>', (e.g,/usr/lib/help/zz),
 *      with the search key being <remainder of arg>, (e.g., 32).
 *      If the argument is all numeric, or if the file as
 *      determined above does not exist, the search will be attempted on
 *      '/usr/lib/sccs.hf', which is the old help file, with the
 *      search key being the entire argument.
 *      In no case will more than one search per argument be performed.
 *
 *      File is formatted as follows:
 *
 *              * comment
 *              * comment
 *              -str1
 *              -str2
 *              text1
 *              text2
 *              -str3
 *              text3
 *              * comment
 *              text4
 *
 *      The "str?" that matches the key is found and
 *      the following text lines are printed.
 *
 *      If the argument is omitted, the program requests it.
 */

char    Oldfile[]       "/usr/lib/sccs.hf";
char    Helpdir[]       "/usr/lib/help/";
FILE    *Iop;


main(argc, argv)
int argc;
char *argv[];
{
        register int i;
	/*
	 *	Tell 'fatal' to issue messages, clean up,
	 *	and return to the setjmp(Fjmp) location.
	 */
	Fflags = FTLMSG | FTLCLN | FTLJMP;

	if (argc == 1)
		findprt(ask());		/* ask user for argument */
	else
		for (i = 1; i < argc; i++)
			findprt(argv[i]);

	exit(0);
}


findprt(p)
char *p;
{
	register char *q;
	char hfile[64];
	char line[512];
	char key[KEYSIZE];

	if (setjmp(Fjmp))			/* set up to return here from */
		return;			/* 'fatal' and return to 'main' */
	if (size(p) > KEYSIZE)
		fatal("argument too long (he2)");
	q = p;
	while (*q && !numeric(*q))
		q++;
	if (*q == '\0') {			/* all alphabetics */
		copy(p, key);
		cat(hfile, Helpdir, "cmds", 0);
		if (!exists(hfile))
			copy(Oldfile, hfile);
	}
	else if (q == p) {			/* first char numeric */
		copy(p, key);
		copy(Oldfile, hfile);
	}
	else {				/* first char alpha, then numeric */
		copy(p, key);		/* key used as temporary */
		*(key + (q - p)) = '\0';
		cat(hfile, Helpdir, key, 0);
		copy(q, key);
		if (!exists(hfile)) {
			copy(p, key);
			copy(Oldfile, hfile);
		}
	}
	Iop = xfopen(hfile, 0);
	/*
	 *	Now read file, looking for key.
	 */
	while ((q = fgets(line, sizeof(line), Iop)) != NULL) {
			repl(line, '\n', '\0');		/* replace newline char */
			if (line[0] == '-' && equal(&line[1], key))
				break;
		}
		if (q == NULL) {		/* endfile? */
			printf("\n");
			fatal(sprintf(Error, "%s not found (he1)", p));
		}
		printf("\n%s:\n", p);
		while (fgets(line, sizeof(line), Iop) != NULL && line[0] != '-')
			if (line[0] != '*')
				printf("%s", line);
		fclose(Iop);
}


ask()
{
		static char resp[KEYSIZE + 1];

		printf("msg number or comd name? ");
		fgets(resp, KEYSIZE + 1, stdin);
		return(repl(resp, '\n', '\0'));
}


clean_up()
{
		fclose(Iop);
}
```

ATTACHMENT III

Inter-UNIX Portability

Thomas L. Lyon[*]

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The availability of the UNIX system on multiple types of computers increases the importance of writing machine independent C programs. Guidelines for portability are given which were formulated through the author's experiences in porting large amounts of software to UNIX on the Interdata 8/32. Statistics are presented which outline the advantages and disadvantages of this conversion to portable code.

September 16, 1977

[*] Work performed while a summer employee at Bell Labs. Presently at Princeton University.

Inter-UNIX Portability

Thomas L. Lyon[*]

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

During the course of moving UNIX to the Interdata 8/32, much has been learned about what is necessary for a C program to be machine independent. This author's definition of a machine independent program is that a single copy of the source for the program can be compiled (with possibly different header files) to execute on any machine running a UNIX system. Note that this last condition is crucial; it eliminates the need to deal with incompatibilities among various operating systems. Experience has shown that operating system incompatibilities are often harder to cope with than hardware differences. The basis of this report is knowledge gained by the author in converting substantial quantities of UNIX software to run on both the PDP-11 and the Interdata 8/32. Included were most important maintenance and user commands, and many library routines; over 19000 lines of C source were converted. This paper will give details of problems encountered in porting programs, guidelines for writing or rewriting programs to be portable, and some statistics about the advantages and disadvantages of portable programming.

Problem Areas

1. Word Length

Perhaps the most obvious difference between machines is in the number of bits in a word. Specifically, the PDP-11 has 16 bit words, while the Interdata has 32 bit words. In practice, programs which depend on word length have been easy to recognize and easy to fix. Word length problems are most often manifested in programs which play with bits, such as those using bit maps or those using bit-wise operators, as in

```
x & 0177770
```

which will clear the low order 3 bits on 16 bit machines, but will clear others as well on machines with larger words. A better approach would be

```
x & ~07
```

Many applications could get rid of this clumsiness altogether by using the bit field feature of C, although even this must be used carefully (see section 7).

[*] Work performed while a summer employee at Bell Labs. Presently at Princeton University.

2. Signed vs. Unsigned Characters

The PDP-11 has the peculiarity that the high order bit of characters is treated as a sign bit, whereas most other machines, including the Interdata, allow only positive values for characters. The most common portability problem involving this difference occurs with a program segment such as

```
char c;
if((c=getchar()) == -1) ...
```

in which the comparison for −1 will fail on any machine but the PDP-11 since characters must be positive. The variable *c* should have been declared as an **int**, because *getchar* is actually an integer valued function; it can return the value −1 as well as any character value.

3. Unsigned vs. Signed Pointer Comparisons

On the PDP-11, pointers are treated as unsigned quantities when they are compared, but on the Interdata, they are treated as signed. This would not cause portability problems if pointers were only generated from valid addresses, but can (and has) lead to difficulties otherwise. For example, one can assign −1 to a pointer on the PDP-11 and then use it as the maximum possible pointer value, but on the Interdata this pointer would be less than all valid pointers. The solution to this problem is to use only valid pointers instead of creating invalid ones.

4. Pointer Wrap-around

It is possible in C that high order bits may be lost during a pointer computation. Although this problem has never occurred in moving from the PDP-11 to the Interdata, it has happened in moving from the PDP-11 to another machine and operating system. A typical program segment vulnerable to this problem is

```
struct a x[100], *p;
for(p=x; p < &x[100]; p++) ...
```

Since the object x[100] does not actually exist, wrap-around could occur in computing its address so that &x[100] was actually less than &x[0]. This particular problem would only occur if the array *x* were at the end of memory; it is much more important to worry about programs which use subscripts which are negative or clearly out of range.

5. Floating Point

Since almost every machine has its own style of floating point, it is not surprising that this could be a problem area in portability. Fortunately, however, about the only problem that arises is that of tolerances: the results of floating computations may be equal on one machine but only almost equal on another machine. In practice, this would seldom arise because defensive programming seems to be the rule in floating comparisons.

6. Shifting

Machine anomalies crop up in right shifting. On the PDP-11, **int** quantities are shifted arithmetically, but on the Interdata they are shifted logically. However, **unsigned** quantities are guaranteed by the C language definition to be shifted logically, so if there is any possibility of trouble, **int** values should be cast to **unsigned** before shifting.

Another problem with shifting is that the maximum shift count varies from machine to machine. Thus, 1L<<32 has the value 0 on the PDP-11 since the 1 bit was shifted into oblivion, but on the Interdata, the result is 1 because only the low order 5 bits of the shift value are used, so no shift takes place.

7. Byte Order

The order of bytes within a word is the problem which has created the most trouble in moving from the PDP-11 to the Interdata 8/32. On the PDP-11, the byte with the lower address is the low-order byte of the word, and the byte with the higher address is the high-order byte of the word. On the Interdata, the reverse is true: the lowest address byte is the highest order byte of a word and the highest address byte is the lowest order byte of a word. This problem most often shows up in communication between the two machines, but can also appear in single machine situations.

Consider the following familiar function.

```
putchar(c)
{
        write(1, &c, 1);
}
```

Since the variable *c* is not declared, it is assumed to be an **int**. Thus, the second argument passed to *write* is the address of an integer, which on the PDP-11 is also the address of the character contained in that integer. But on the Interdata, the address passed is that of the high order byte of the integer rather than that of the low order byte which is where the character is. Thus, only null characters would ever be written. This particular example would work if *c* were declared as a **char.**

A related problem is that the ordering of bit fields within a word is the same as the ordering of bytes. So, on the PDP-11, the first field is in the lowest order bits of the word; on the Interdata, the first field is in the highest order bits. This problem should affect only those programs which depend on the relative positions of bit fields (unfortunately, most do).

The byte-ordering differences create very serious problems in machine to machine communication. Because of the nature of I/O, all communication between the PDP-11 and the Interdata 8/32 is in a byte-by-byte fashion. Thus, if one transmits character data to the other machine, that data is reconstructed in the other machine in the proper order. However, if one sends integer data, it is first broken down into byte-by-byte order and when the data is reconstructed on the other machine the bytes of the integer are in the wrong order. There does not exist any easy solution to this problem. If one must transmit integer data from one machine to another, the exact format of that data must be known so that the appropriate bytes may be swapped either before or after transmission.

Portability Guidelines

During the course of the UNIX portability project, it has become apparent that the best criterion for ease of portability is that programs should be written in good style. "Good style" is admittedly hard to define, and this definition may have been influenced by experiences in portability; nonetheless, it has been this author's overwhelming experience that a stylistically pleasing program is an easily portable program. With regard to this, the following sections will discuss techniques which will vary from being necessary for machine independence to being suggestions for better style.

1. Use Header Files

One of the most important steps in writing a portable program is to use the C preprocessor features of include files and definitions. Whenever definitions of data types, formats, or values are shared by multiple programs, include files should be used to centralize these definitions and to assure that there is only one copy to change if change is ever needed. The directory /usr/include contains include files intended for public use. A line of the form "#include <xxx.h>" will include the file /usr/include/xxx.h. Typical of these files are <dir.h>, a definition of the structure of a directory, and <signal.h>, a definition of system signal numbers.

Another important use of header files is to isolate environment dependent data, such as default file names or options. In fact, anything that could vary between systems, even if they are on the same type of computer, should be placed in a header file where it may be easily located and modified.

2. Isolate Constants

With few exceptions, whenever a constant appears in the body of a program one can be justifiably worried about that program's portability or style. Innocent looking constants like 0, 1, and 2 are sometimes exceptions, but if one found a 79 in a program one would have to stop and ask what that 79 really means, how it was derived, and why it isn't expressed in terms which make its meaning obvious. For instance, programs often maintain state variables which take on a number of values depending on the state of the program. These values are best expressed by using the #define feature of the preprocessor to associate names with the values, instead of using numeric constants in the code which have no immediate meaning.

Often a constant is used to represent the size of some object; such usage is not at all portable and should be changed to use the **sizeof** operator of C. If a program uses constants as array subscripts, then it is likely that each element of that array has some different type of meaning. Such array uses could best be replaced by structures, which not only have a different meaning for each element, but also have a different name for each element to make that meaning apparent. The most common examples of this misuse of arrays are in the stty and gtty system calls, which, on the PDP-11, are commonly invoked with a three word array as an argument, instead of the structure defined in /usr/include/sgtty.h. This usage is also non-portable since the structure is only two words on the Interdata. Another type of constant to watch out for is that of constant file names. Such file names are almost always environment dependent and should be isolated in a header file or #define statement.

3. Avoid Efficiency Tricks

Occasional problems encountered in porting a program are due to efficiency tricks, i.e., places where the programmer has taken advantage of his knowledge of the machine's architecture to give a slight efficiency boost to his program. Such constructs are almost always non-portable, and should be replaced by constructs which are guaranteed by the C language definition to work on any machine. Indeed, it could be argued that a C programmer should never know what machine his program will run on, so that he writes code strictly by the C manual instead of by the machine manual.

4. Use New C Features

Perhaps the greatest help in writing portable C programs is given by some of the newer features of the language. After studying the needs of portability, these features were chosen to allow ease of expression and portability in what are normally messy areas. The **typedef** feature allows the programmer to isolate the definition of machine dependent data types, while the **sizeof(type)** construct allows finding the size of any of these defined data types. The **union** declaration allows the overlaying of data, which has historically been a very trying problem in portability.

5. Use Canned Routines

A wise rule to follow in programming is: don't re-invent the wheel. One should use publicly available canned routines as much as possible; this reduces the possibility of duplicated source code, leads to increased modularization, makes programming easier, and makes maintenance of any particular function easier.

It is not at all uncommon to see programs containing functions to do exactly the same thing as publicly available functions; *atoi* and *strcmp* are prime examples. The programmer should take it upon himself to be acquainted with recent additions to the C library. The most important package of routines as far as portability is concerned is the standard I/O library, developed as a replacement and enhancement to the portable I/O library. The *getc* and *putc* macros provided by this package should be used to replace the old *getc* and *putc* routines of the PDP-11, since the old routines do not exist for the Interdata 8/32. Routines which have proven particularly useful in revision of programs to remove duplicated function include *sprintf, fgets,* and *fputs.*

6. Use Lint

The *lint* command, a C program checker, has proved to be a valuable aid to porting programs. Used with the strict type checking option (-s flag), *lint* provides very useful information about possibly non-portable constructs. If one were to use and respect *lint* at every stage of a program's development, that program would be very easily portable to another UNIX system; there would be a good chance that no changes would be needed to port it. When converting an old program into a portable program, output from *lint* is very useful in identifying trouble spots. On the other hand, the output from *lint* is often extremely voluminous; rethinking the data structures often does wonders to shut up complaints about strict type checking.

Statistics

This section will give some estimate of the cost of writing portable C programs. Of about 220 programs in the /usr/src/cmd directory, over 80 have been converted into portable programs which will run on either the PDP-11 or the Interdata 8/32. This represents about 19000 of 45000 lines of C. In general, it has been observed that when a program was converted into a portable program, the size of the source code decreased and the size of the object code increased.

The average decrease in source size was 9%, while the total amount of source also decreased by 9%. This decrease was caused by the use of more canned routines and header files, as suggested in the previous section. The source size is defined as the number of lines in the program.

The average increase in object size was 56%; total object size increased by 20%. Object size is defined as the total given by the *size* program. This isn't as bad as it seems. The huge average increase is due to the use of the standard I/O library in otherwise small programs. For instance, the *echo* program grew by a whopping 880%, but this is actually only 2078 bytes. The total increase of 20% is a true cost of the style and portability improvements; this cost, however, is completely cancelled by the gain in ease of maintenance. Having only one copy of a program's source prevents major administrative headaches, and the improvements in style make understanding and modifying programs much easier. In addition, the 2½ months of effort involved in this conversion are a one-time cost; any future moves of UNIX to still other machines will not need to duplicate this work.

The appendix gives a program by program breakdown of sizes before and after conversion to portable code.

Appendix

This appendix gives new and old, source and object sizes for a number of individual programs. The programs shown are those for which it was easy to collect statistics; these programs are not all that were converted, but are representative of those that were.

Size of Source Code (lines)

Command	Old Size	New Size	% Diff
ac	245	243	-0.82
ar	617	620	0.49
chown	95	54	-43.16
clri	77	79	2.60
cmp	125	120	-4.00
comm	180	167	-7.22
cron	242	241	-0.41
date	201	152	-24.38
dcheck	249	220	-11.65
dd	536	533	-0.56
df	104	97	-6.73
diff	664	636	-4.22
echo	32	21	-34.38
getty	248	220	-11.29
ind	23	25	8.70
init	299	295	-1.34
kill	38	40	5.26
mail	429	338	-21.21
mesg	57	52	-8.77
mkfs	589	583	-1.02
mount	64	65	1.56
mv	155	184	18.71
ncheck	358	328	-8.38
newgrp	159	55	-65.41
nice	40	43	7.50
od	628	239	-61.94
passwd	197	156	-20.81
pr	442	414	-6.33
rev	44	44	0.00
rm	87	118	35.63
size	43	46	6.98
sort	813	823	1.23
split	88	81	-7.95
strip	117	111	-5.13
stty	322	284	-11.80
su	123	38	-69.11
tabs	265	246	-7.17
tail	162	161	-0.62
tee	63	64	1.59
test	141	136	-3.55
time	129	93	-27.91
tr	144	144	0.00
uniq	173	141	-18.50
wall	70	66	-5.71
who	88	51	-42.05
write	176	181	2.84

Size of Object Code (total bytes)

Command	Old Size	New Size	% Diff
ac	10154	10258	1.02
ar	8492	10136	19.36
chmod	2428	3968	63.43
chown	3634	6028	65.88
clri	2854	2854	0.00
cmp	2950	4704	59.46
comm	5362	5274	-1.64
cron	3620	6122	69.12
date	2640	2852	8.03
dcheck	11888	11972	0.71
dd	5108	7294	42.80
df	2156	4362	102.32
diff	9320	9616	3.18
echo	236	2314	880.51
getty	1170	1198	2.39
ind	1612	3650	126.43
init	3620	3612	-0.22
kill	2350	2350	0.00
mail	10450	10746	2.83
mesg	1080	3884	259.63
mkfs	11024	11000	-0.22
mount	3886	5490	41.28
mv	3336	5602	67.93
ncheck	41306	41370	0.15
newgrp	8542	10594	24.02
nice	2650	4714	77.89
od	5278	6104	15.65
passwd	6476	10194	57.41
pr	12204	15632	28.09
rev	4100	4100	0.00
rm	4148	4452	7.33
size	4100	4128	0.68
sort	10222	10256	0.33
split	2252	4344	92.90
strip	2968	2978	0.34
stty	2498	4794	91.91
su	6694	8504	27.04
tabs	1758	3346	90.33
tail	5730	5728	-0.03
tee	1960	1960	0.00
test	1420	4538	219.58
tr	4240	4282	0.99
uniq	5012	6622	32.12
wall	6108	8478	38.80
who	5962	5962	0.00
write	5966	5984	0.30